Spi2Java User Manual - Version 3.1

Alfredo Pironti, Davide Pozza, Riccardo Sisto

Politecnico di Torino
Dipartimento di Automatica e Informatica
C.so Duca degli Abruzzi, 24
10129 - Torino - Italy

Disclaimer: Please consider that the Spi2Java framework is a research prototype. It has not undergone all testing and validation that is usually expected for a production environment software. We are already conscious of some of its deficiencies and possible improvements. Please feel free to contact us by email if you encounter or find any problem. Feedbacks and suggestions are very appreciated. When you contact us, please write to all of us

`{alfredo.pironti,riccardo.sisto}@polito.it`

and use "Spi2Java" as a prefix in the subject. We will answer as soon as possible.

Contents

1 Introduction

Formal methods have the potential to significantly improve the trustworthiness of critical software, especially when development turns out to be intrinsically error prone, and bugs difficult to discover, as it happens with cryptographic protocols. However, one of the problems that still limit the widespread use of formal methods is the high level of expertise that they normally require and the high cost of development that their use implies. A way to partially overcome the above problem and to improve the acceptance and productivity of formal methods is to provide methodologies and tools that simplify the use of formal methods, introducing automation and hiding underlying complexity.

Spi2Java is a set of tools that support a model-based approach to cryptographic protocol implementation based on the spi calculus [1]. By using these tools it is possible to take advantage of the capabilities of high-level formal analysis tools such as ProVerif[2] and S^3A[3], while at the same time ensuring the correspondence between high level formal models and their implementations.

In short, the main Spi2Java tools and their aims are as follows:

spi calculus parser is a pre-processor and parser for the spi calculus description of the protocol.

spi2java refiner is a type checker that infers Java types for the spi calculus terms used in the (untyped) protocol description.

spi2java code generator is an automatic code generator that emits the Java code implementing the protocol described in spi calculus.

The rest of this document is organized as follows. Section 2 specifies system requirements for running the Spi2Java framework and explains how to get started with it. Section 3 introduces the spi calculus language with the syntax accepted by the tools and presents a reference example that will be used throughout this document. Section 4 shows in detail, mainly through the full development of the reference example, how the tools can be used to produce an implementation of a cryptographic protocol. Section 5 explains how to use some correlated auxiliary tools.

2 Getting Started with Spi2Java

2.1 Prerequisites

The Spi2Java tools are entirely implemented in Java. For this reason, a Java Runtime Environment version 1.6.x or later is required to run the tools composing the framework.

An Apache Ant version 1.7.x or later *build.xml* script is provided, in order to compile the tools from sources or to perform other distribution-related tasks. However, Apache Ant (and the Java SDK) is not needed for the typical user, as the distribution includes ready-to-use executable jar files.

2.2 Getting the Software and Setting It Up

You should have downloaded a copy of the framework, which includes this user manual, from the Spi2Java web site, at http://spi2java.polito.it/

The framework comes in a compressed archive, either in `tar.gz` or in `zip` format. Whatever archive you downloaded, extract it; it will create a new directory containing the extracted content.

Please take a look at the `README` file: it contains general information on the framework, and a description of the files included in the distribution.

No other installation steps are required. In particular, as this user manual is concerned, the `jars` directory contains all pre-compiled jars. Those containing a `tui` in their name are the executable jars, providing a "textual user interface" (also known as cli – "command line interface") to the Spi2Java tools. In general, on a UNIX platform, assuming that the `JARS` environment variable points to the `jars` directory of the Spi2Java framework, you can run one of the tools by invoking the following command from the command line:

```
java -jar $JARS/<jar_filename.jar> [options]
```

All programs support a `-h` option that prints an help screen and exits, and a `-debug` option that enables debugging mode, with extensive error reporting, useful for troubleshooting or bug reporting.

3 Spi Calculus and Reference Example

The spi calculus is defined in [1] as an extension of the π calculus [4] with cryptographic primitives. It is a process algebraic language designed for describing and analyzing cryptographic protocols. These protocols heavily rely on cryptography and on message exchange through communication channels; accordingly, the Spi-Calculus provides powerful primitives to express cryptography and communication.

This section illustrates the spi calculus syntax accepted by the framework and describes the language semantics informally.

The language used by Spi2Java is basically the spi calculus as defined in [1] with some extensions. The syntax has been slightly modified with respect to [1], in order to make it machine-readable

A spi calculus specification is a system of independent processes, executing in parallel; they synchronize via message-passing through named communication channels. The spi calculus has two basic language elements: terms, to represent data, and processes, to represent behaviors.

3.1 Term Syntax

Terms can be either atomic elements, i.e. names, including the special name 0 representing the integer constant zero, and variables, or compound terms built using the term composition operators listed in Tab. 1. Names can represent for example communication channels, atomic keys, key pairs, nonces (also called *fresh names*) and any other unstructured data.

The informal meaning of the term composition operators is as follows:

- (σ, ρ) is the *pairing* of σ and ρ. It is a compound term whose components are σ and ρ. Pairs can always be freely split into their components.
- $\mathrm{suc}(\sigma)$ is the *successor* of σ. This operator has been introduced mainly to represent successors over integers, but it can be used, more generally, as the logical successor of any term.

σ, ρ ::=	terms
m	name
x	variable
(σ, ρ)	pair
0	zero
suc(σ)	successor
H(σ)	hashing
$\sigma\sim$	shared-key
$\{\sigma\}\rho$	shared-key encryption
$\sigma+$	public part
$\sigma-$	private part
$\{[\sigma]\}\rho$	public-key encryption
$[\{\sigma\}]\rho$	private-key encryption (signature)

Table 1: Term syntax of Spi-Calculus

- H(σ) is the *hashing* of σ. H(σ) represents a function of σ that cannot be inverted.
- Term $\sigma\sim$ represents a shared key obtained by some key material σ.
- Term $\{\sigma\}\rho$ is the ciphertext obtained by encrypting σ under key ρ using a shared-key cryptosystem.
- $\sigma+$ and $\sigma-$ represent respectively the public and private half of a key pair σ. $\sigma+$ cannot be deduced from $\sigma-$ and vice versa.
- $\{[\sigma]\}\rho$ is the result of the public-key encryption of σ with ρ.
- $[\{\sigma\}]\rho$ is the result of the signature (private key encryption) of σ with the private key ρ.

In addition to the basic term composition operators shown in Tab. 1, some syntactic shortcuts are available for common combinations of term compositions:

- $(\sigma_1, \sigma_2, \cdots, \sigma_n)$ is a shortcut for the left-associated nested pairs $(\cdots((\sigma_1, \sigma_2), \sigma_3), \cdots), \sigma_n)$.
- any unary operator can be applied to a sequence of terms, with the meaning that it is applied to the corresponding nested pair. For example, $H(\sigma_1, \sigma_2, \sigma_3)$ stands for $H((\sigma_1, \sigma_2, \sigma_3))$, which in turn stands for $H(((\sigma_1, \sigma_2), \sigma_3))$

3.2 Process Syntax

Besides term specification, spi calculus also offers a set of operators to build behavior expressions which formally specify the behavior of processes.

Tab. 2 shows the operators available to build behavior expressions. Their informal meaning is:

- $\sigma\langle\rho\rangle.P$ is an *output process*, ready to output term ρ on the channel represented by term σ when a synchronization occurs. The behavior after the synchronization is described by behavior expression P.

$P,\ Q$ ::=	process behavior expressions
$\sigma\langle\rho\rangle.P$	output
$\sigma(x).P$	input
$P \mid Q$	parallel composition
$(@\,m)\ P$	restriction
$!P$	replication
0	nil
$[\sigma\ is\ \rho]\ P$	match
$[\sigma\ is\ \rho]\ (P)\ else\ (Q)$	
$let\ (x,\ y)\ =\ \sigma\ in\ P$	pair splitting
$let\ (x,\ y)\ =\ \sigma\ in\ (P)\ else\ (Q)$	
$case\ \sigma\ of\ 0 : P\ \mathrm{suc}(x) : Q$	integer case
$case\ \sigma\ of\ \{x\}\rho\ in\ P$	shared-key decryption
$case\ \sigma\ of\ \{x\}\rho\ in\ (P)\ else\ (Q)$	
$case\ \sigma\ of\ \{[x]\}\rho\ in\ P$	private-key decryption
$case\ \sigma\ of\ \{[x]\}\rho\ in\ (P)\ else\ (Q)$	
$case\ \sigma\ of\ [\{x\}]\rho\ in\ P$	public-key decryption
$case\ \sigma\ of\ [\{x\}]\rho\ in\ (P)\ else\ (Q)$	
$check\ \sigma\ of\ \tau\ with\ \rho\ in\ P$	signature check
$check\ \sigma\ of\ \tau\ with\ \rho\ in\ (P)\ else\ (Q)$	

Table 2: Process Syntax of Spi-Calculus

- $\sigma(x).P$ is an *input process*, ready to perform an input from channel σ when a synchronization occurs. The behavior after a synchronization in which the received message is term ρ is described by behavior expression P with any occurrence of x replaced by ρ, which is denoted $P[\rho/x]$.

- $P \mid Q$ is a *parallel composition* where P and Q run in parallel. They may either synchronize with each other or with the external environment separately. This operator is commutative and associative.

- $(@m)P$ is a *restriction* process which makes a fresh, private name m and then behaves as described by P.

- $!P$ is a *replication* where an unbounded number of instances of P run in parallel.

- $[\sigma\ is\ \rho]P$ is a *match* process which behaves as described by P if the terms σ and ρ are the same, and is stuck otherwise. If an *else* Q branch follows, the process behaves as described by Q if the two terms do not match.

- 0 is the *nil* process: it is a stuck process.

- $let\ (x,\ y)\ =\ \sigma\ in\ P$ is a *pair splitting* process. If term σ is a pair $(\rho_1,\ \rho_2)$, this process behaves as $P[\rho_1/x, \rho_2/y]$, otherwise it is stuck or it behaves as Q if an *else* Q branch follows.

- $case\ \sigma\ of\ 0 : P\ \mathrm{suc}(x) : Q$ is an *integer case* process. If σ is 0, it behaves as P; if σ is $\mathrm{suc}(\rho)$, it behaves as $Q[\rho/x]$. It is stuck otherwise.

- $case\ \sigma\ of\ \{x\}\rho\ in\ P$ is a *shared-key decryption* process. If σ is a ciphertext taking the form $\{\eta\}_\rho$, it behaves as $P[\eta/x]$, otherwise it is stuck unless an $else\ Q$ branch follows.

- $case\ \sigma\ of\ \{[x]\}\rho\ in\ P$ is a *private-key decryption* process and behaves as $P[\eta/x]$ if σ is a term η encrypted under a public key whose corresponding private key is ρ. Otherwise it is stuck, unless an $else\ Q$ branch follows.

- $case\ \sigma\ of\ [\{x\}]\rho\ in\ P$ is a *public-key decryption (or signature check)* process and behaves as $P[\eta/x]$ if σ is a term η encrypted under a private key whose corresponding public key is ρ. Otherwise it is stuck, unless an $else\ Q$ branch follows.

- $check\ \sigma\ of\ \tau\ with\ \rho\ in\ P$ checks whether σ is a valid signature of the message τ using public key ρ. If this is the case, then P is executed; otherwise the process is stuck, or it behaves like Q if an $else\ Q$ branch follows.

Some of the above expressions can have an optional *else* branch, representing the behavior that occurs when the operation does not succeed. For example, in expression $case\ \sigma\ of\ \{x\}\rho\ in\ (P)\ else\ (Q)$ process Q represents the behavior that occurs when the decryption of σ with key ρ does not succeed. Note that brackets are mandatory when the *else* branch is defined. Moreover, there is no way to join if/else branches, much like it happens in compositions. When the *else* branch is missing, the process gets stuck in case of unsuccessful completion of the operation. This means that the execution of the protocol stops with an exception. If we indicate unsuccessful termination by $fail$, expressions without the *else* branch can be interpreted as expressions having a default $else\ fail$ suffix. In this respect, the two processes

$$case\ \sigma\ of\ \{x\}\rho\ in\ P$$

and

$$case\ \sigma\ of\ \{x\}\rho\ in\ (P)\ else\ (0)$$

are semantically equivalent from a spi calculus point of view, but result in two different Java implementations. If decryption fails, the first process implicitly terminates with a failure (that is, throwing an exception in Java); while the second process “gets stuck”, that is correctly terminates, thus not throwing an exception, but normally returning to its caller. The user can never invoke a $fail$ process, it is implicitly added by the parser when no *else* branch is specified.

Behavior expressions are used to define processes. A process definition takes the syntax

$$Procname(x_1, \cdots, x_n) ::= P$$

where *Procname* identifies the process, P is the behavior expression that specifies the process behavior and x_i are the process formal arguments.

In order to facilitate modular definitions of processes and process definition reuse, a behavior expression may include a process instantiation whenever a process is expected. A process instantiation takes the form

$$Procname(\sigma_1, \cdots, \sigma_n)$$

where σ_i are the actual arguments that replace the corresponding formal ones.

3.3 Spi Calculus Source Files Syntax

A spi calculus source file is divided in two sections: constants definition, and processes definitions.

Constants are defined by the keyword `const:` followed by a comma separated list of free names, ended by a dot '`.`'. For example, a valid constants definitions is

```
const: A,B,C.
```

If no constants are to be declared, this section can be omitted. The spi calculus parser of spi2java checks that any declared constant is used at least once in the specification. Conversely, for each process P, the parser checks that all of its free names are declared in the process formal parameters, or in the `const:` section.

The following, mandatory section, is a collection of process definitions. Processes cannot be recursive (not even indirectly). There is no semantic difference between constant data, and formal parameters of a process. In fact, constant data can be regarded as implicit formal parameters that are automatically added by the parser to process definitions, when necessary. As a general rule, it is recommended to put constant data of the protocol in the `const:` section, so that all processes can use them, without the need to declare them in the formal parameters list; this may include, for example, identifiers of protocol messages, constant strings, or enumerations. Session specific values, such as session identifiers, should be put in the processes formal parameters lists.

A *top level* process is a process that is not instantiated by any other process. A specification can contain more than one top level process.

Comments are supported in C style. They can begin with a `//` and extend until the end of the line; or they can be enclosed within `/*` and `*/`, spanning on multiple lines (like in C, this form of comment cannot be nested).

3.4 An Example

Spi2Java comes with a full example that can be used to get acquainted with the framework. All required steps to get a working implementation from the spi calculus specification are detailed in this manual. The relevant data are placed in `data/full_example`. From now on, if not otherwise stated, we will assume that this directory is the base for any referenced file or issued command.

The chosen protocol is the key exchange one presented in section 6.1.5 of [5]. Since this protocol has no specific name, we will call it "sof", standing for "six (dot) one (dot) five".

The `spi/sof.spi` file contains a possible full spi calculus specification of sof. To make it more understandable, figure 1 shows an alternative representation of that content.

The `const:` section contains the declaration of the A and B agents identities, as well as the key store GET operand (more on this later). Note that in this example, A and B identities are fixed. This means that this specification only handles the case where two known actors are interacting, other third party protocol actors are not taken into account. The goal of this example is to show the usage of spi2java, rather than implementing a real security protocol indeed.

Constant data section
`const:  A, B, KEYSTORE_GET.`

Line	sofA Actor	sofB Actor	Meaning
1	`sofA(A,cAB,cKS) :=`	`sofB(B,cAB,cKS) :=`	
2		`(@Rb)`	
3	`cAB((xB,xRb)).`	`cAB<(B,Rb)>.`	`B → A: B, Rb`
4	`cKS<(KEYSTORE_GET,(A,xB))>.`		
5	`cKS(Kab).`		
6	`(@Ra)`		
7	`(@K)`		
8	`cAB<(A,{H(xRb),Ra,A,K~}Kab)>.`	`cAB((xA,encData)).`	`A → B: A, {H(Rb),Ra,A,K}Kab`
9		`cKS<(KEYSTORE_GET,(xA,B))>.`	
10		`cKS(Kab).`	
11		`case encData of {plainText}Kab in`	
12		`let (hashOfRb,xRa,xA,K) = plainText in`	
13		`[ hashOfRb is H(Rb) ]`	
14	`cAB((xB,encData)).`	`cAB<(B,{H(xRa)}K)>.`	`B → A: B, {H(Ra)}K`
15	`case encData of {hashOfRa}K~ in`		
16	`[ hashOfRa is H(Ra) ]`		
17	`0`	`0`	

```
Key Store Process
keyStore(cKS,Kab) :=
  cKS((request,params)).
  [ request is KEYSTORE_GET]
  [ params is (A,B) ]
  cKS<(Kab)>.
  0
```

```
Protocol Sessions Instantiation
Inst(cAB) :=
  (@Kab) // generate secret shared key
  (
    // processes A
    /*![10]*/(
      (@cKS) // create secret keystore channel
      (
        keyStore(cKS,Kab~) // run keystore instance for A
      |
        sofA(A,cAB,cKS) // run A instance
      )
    )
  |
    // processes B
    /*![10]*/(
      (@cKS) // create secret keystore channel
      (
        keyStore(cKS,Kab~) // run keystore instance for B
      |
        sofB(B,cAB,cKS) // run B instance
      )
    )
  )
```

Figure 1: A possible spi calculus specification of the “sof” protocol.

Then, the specifications for the A and B agents are given. The “Line” column enumerate the specification lines, while the “sofA Actor” and “sofB Actor” columns contain the specifications of the A and B agents respectively. The “Meaning” column provides an informal, intuitive representation often encountered in the literature, where $A \to B : \sigma$ means that A sends message σ to

B.

The sof protocol subsumes that a long-term key is pre-shared between the protocol participants. At the implementation level, one way to achieve this is to use a key store. Each participant has its own key store, which associates a key with the participants identities. A simple key store modeling strategy is to represent the key store as a separate process that interacts with the corresponding protocol principal through a dedicated communication channel (the *key store channel*) not accessible by the intruder. The operations of getting and storing a key are modeled as pairs of inputs/outputs on the key store channel. For example, at line 4, the sofA process sends the message `(KEYSTORE_GET,(A,xB))` on the `cKS` channel. By the `KEYSTORE_GET` *operand*, sofA is asking the key store to retrieve the key associated with the `(A,xB)` identifier; being `(A,xB)` a pair, the resulting identifier will be the concatenation of the `A` and `xB` identifiers. At line 5, sofA reads back the retrieved key from the key store. If they key could not be found, then the process would be stuck. More details about key store interaction can be found in section 4.3.1, when describing the key store channel.

In a run of the sof protocol, three messages are exchanged through a public communication channel named `cAB`:

1. At line 3 `B` sends to `A` a Pair composed of its identifier `B` and a nonce `Rb`, created at line 2. When `A` receives the message, it retrieves the key `Kab` shared with `B` from its local key store (lines 4–5). If key retrieval is successful (line 6), `A` generates its nonce `Ra` (line 7), and some fresh key material `K` that will be used for the generated session key (line 8).

2. Then at line 9 `A` sends a pair to `B`. The left part of the pair is `A`'s identity; the right part is the encryption under the pre-shared key `Kab` of the hash of `B`'s nonce, as received in message one (stored locally in variable `xRb`), `A`'s nonce `Ra`, `A`'s identity, and finally the session symmetric key `K~`. When `B` receives the message, it uses `A`'s provided identifier to retrieve the pre-shared key `Kab` from the local key store (lines 10–11). If key retrieval is successful at line 12, the received data are decrypted (line 13) and split into their parts (line 14).

3. If at line 15 the received term `hashOfRb` matches the locally reconstructed `H(Rb)`, `B` sends the last protocol message to `A` and terminates. This message contains `B`'s identity again, followed by the hash of the received `A`'s nonce, encrypted under the session key `K`. Note that in `A`, the key material is a fresh name called `K`, and the shared key built upon this material takes the form `K~`. Instead, in `B`, `K` is a variable that is supposed to be bound to the shared key sent by `A`.

   When `A` receives the message, at line 17 it tries to decrypt it using the session key `K~`, and checks that the plaintext matches `H(Ra)` (line 18). If all checks are successful, `A` correctly terminates.

In this simple abstraction of the key store, the `keyStore` process just waits for a request. When a request comes, if the operand is `KEYSTORE_GET` and the identifier matches the concatenation of `A` and `B` constant identities, then the pre-shared key is returned; else the process becomes stuck, effectively not returning any key. It is straightforward to enhance the key store to handle more than one key/identifier pair.

The `Inst` process puts all pieces together, by generating the pre-shared key between `A` and `B`, and running several concurrent instances of `A` (and its key store) and `B` (and its key store). Note that this version of Spi2Java does not support replication, this is why the replication operator `![n]` is commented out in the specification of `Inst`. In future versions of Spi2Java, replication is going to be supported. The `n` parameter in the `![n]` operator shall be an integer indicating the number of replicated processes, in case of bounded replication. Not currently supporting replication is not a limitation during code generation, because in fact only the `sofA` and `sofB` processes will be used. Very often indeed, processes describing protocol actors are simple sequential specifications, so no composition nor replication are usually required. Not supporting replication in Spi2Java is not a limitation in the formal verification phase too. Assuming the verification tool supports the same syntax as Spi2Java, it is enough to uncomment the replication operators to get the complete behavior.

Note that each agent runs in parallel with its key store, and each agent-key store pair has its own dedicated channel. Moreover, if present, replication would be put outside the agent-key store composition, and not, for example, close to each agent and to each key store. This model ensures that in each protocol session, each agent can only communicate with its own key store, and not with key stores of other sessions.

Also note how spi calculus enables the precise specification of all the operations performed by the protocol participants, including their interactions with their local key stores, though at an abstract level (no information about message encodings or encryption algorithms is specified).

Of course, this specification would not be complete if cryptographic details were not given. Although they are not needed for the spi calculus specification, we provide them here for completeness. They will be referenced later when implementing the protocol actors.

We prescribe that the pre-shared cryptography must use the 3DES algorithm, with the standard key strength of 192 bits; while the session key must be a DES key, with the standard key strength of 56 bits. All nonces must be 8 bytes long.

4 Spi2Java Development Methodology and Tool Support

This section describes the Spi2Java approach to model-based development of cryptographic protocols. The approach is illustrated by the data flow diagram in figure 2.

The reference example introduced in the previous section will be used for explaining and exemplifying the approach and the use of the Spi2Java tools. Moreover, in [6] the interested reader can find a case study, where the spi2java framework has been used to automatically generate an SSH Transport Layer Protocol client implementation.

4.1 Writing and Compiling the Formal Model

The programmer normally starts from an informal description of the protocol (e.g. an Internet RFC or other standard document), and manually derives a

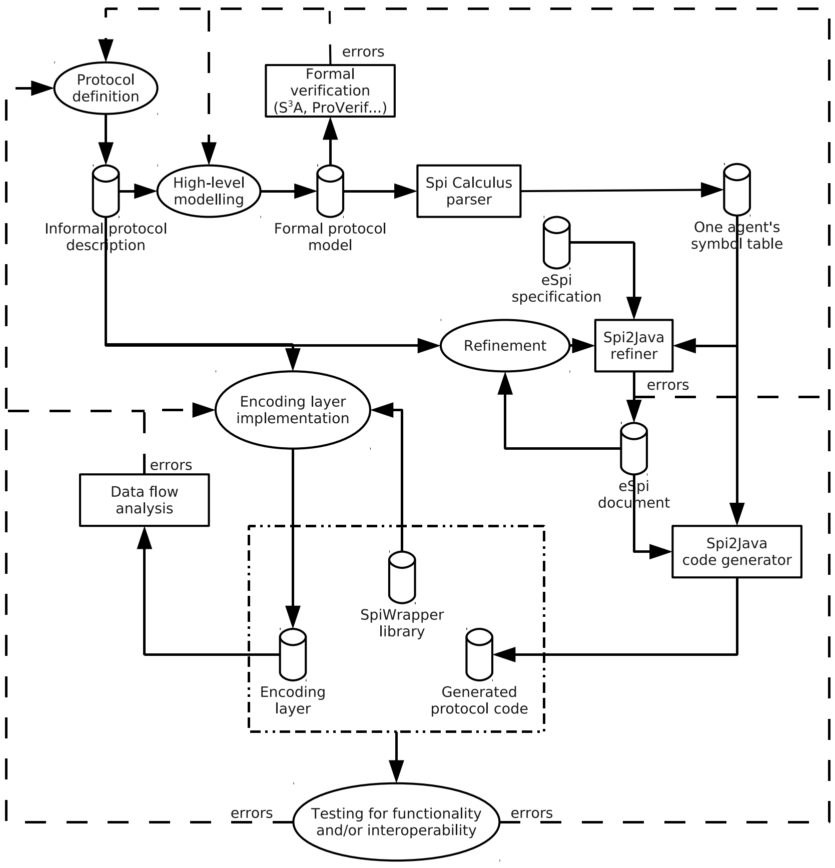

Figure 2: A data flow diagram of the suggested model-based development technique.

formal spi calculus model that only captures the high-level logic of the protocol in an abstract way. For the reference example, the informal protocol description, without low-level details, and the formal spi calculus model have already been described in the previous section, and is available in the `spi/sof.spi` file.

By using spi calculus parser tool, included in the jar *spiParser.tui.jar*, the syntax of the descriptions can be validated, and a symbol table for each role is produced.

Symbol table for actor A can be generated by issuing the following command.

```
java -jar $JARS/spiParser.tui.jar -i spi/sof.spi -p sofA
          -o spi/sofA/sofA.css -r spi/sofA/sofA.rev
```

Please remember that the `$JARS` environment variable points to the location of the jars distributed with the Spi2Java framework, and that all command are executed from the base `data/full_example` directory. The `-i` option indicates the input source file, which is the file containing the full spi calculus specification, as already explained. The `-p` option asks to extract a symbol table for the "sofA" process only; we will generate Java code implementing only one actor at a time, not the whole protocol description. The `-o` option indicates the symbol table output file. The extension `css` stands for Canonical Spi Specification, and is the internal format used by the Spi2Java tools. Finally, note that the Spi2Java parser translates any syntactic shortcut into its low level expansion (e.g. term sequences are translated to nested pairs, and the $else\ fail$ suffix is added to any expression supporting the else branch). The optional `-r` switch saves the expanded spi calculus-like version to a file, which may be useful in the refinement phase, since the expanded version of the specification shall be refined.

The same steps can be repeated for the “sofB” actor, updating the command line option arguments accordingly. Specific instructions about the “sofB” actor are given only when they significantly differ from the “sofA” actor.

4.2 Hints for the Formal Verification Step

Once a spi calculus model has been written, it can be analyzed for verifying its logical correctness. Formal verification of protocols is not the goal of the spi2java tools, nor of their user manual. Nevertheless, some hints and support to use the S^3A [3] and proverif [2] formal verification tools are provided.

The input language of S^3A is the original spi calculus as defined in [1], whereas the Spi2Java parser accepts a slightly larger language (e.g. else branches and the shared key constructor operator). Only description that do not use these features can be analyzed by S^3A. Note also that the S^3A native parser does not accept comments and do not support the `const:` declaration section. Nevertheless, the css symbol table created by the Spi2Java parser is compatible with the one used by S^3A. So it is possible to feed the css file produced by the Spi2Java parser directly to S^3A, and this is actually the recommended way: write a spi calculus specification and use the Spi2Java parser to get a symbol table (css file) for the whole protocol, then feed S^3A with the resulting css file.

For example, the command

```
java -jar $JARS/spiParser.tui.jar -i spi/sof.spi
                                  -o spi/sof.css
```

creates a symbol table for the “top level” process (because the `-p` option is not specified), which is the $Inst$ process, actually representing the full protocol execution (modulo replication).

In order to analyze a spi calculus model by ProVerif, a syntax conversion is needed, because proverif uses different syntactic conventions. This operation can be performed by using the provided *Spi2Proverif* tool (see section 5.3). Note that, differently from S^3A, ProVerif can handle all the language features admitted by Spi2Java and even others. In addition, it accepts unbounded replications, which are useful in the instance process for analyzing the protocol behavior with an unbounded number of sessions.

4.3 Refining the Formal Model

4.3.1 Theory

In order to derive a Java application from the spi calculus source, the spi2Java refiner tool can be used to fill the low-level implementation details that are abstracted away by the spi calculus language.

A tool like the spi2Java refiner can automatically infer *some* information about the missing details that are not present in the formal high-level model: the type of certain data can be automatically inferred by looking at how they are used. For example, in the output process $\overline{c}\langle M\rangle.P$, c can be inferred to be a channel. The implementation details that cannot be automatically inferred must be manually provided by the user, who can get them from the informal protocol description.

However, an interesting feature of the tool is the possibility to get early prototyping without any (or very few) manual intervention, just after having written the formal model. In order to get the early prototype, the tool can fill all the missing needed data with default values, which allows to immediately get a complete specification. The user can later change the default values to accommodate needs; after editing, the spi2Java refiner checks the user-given values for correctness and coherence with the reference spi calculus specification.

The low-level implementation details can be grouped into two main categories:

1. Cryptographic and Configuration parameters
2. Encoding/decoding functions (or, simply, encoding functions)

The first group of details specifies parameters such as "what algorithm must be used for a particular encryption operation" or "what network interface must be used by a particular channel". In order to make the generated code compliant with the implemented protocol, it is necessary that these parameters can be set independently, at compile time or at run time, for each data item.

The second group of details deals with the transformation from the internal representation of messages into their external representation, and vice versa. The internal representation is the one used to perform all the operations prescribed by the protocol logic on the data; whereas the external representation is the stream of bytes that must be exchanged with the other parties. Decoupling internal and external representations is necessary in order to obtain interoperability, because the external representation must conform to the agreed binary formats defined for the protocol.

In order to enable agile prototyping, a default encoding/decoding layer, which uses the Java serialization, is provided; however, in real environments, this default encoding/decoding layer normally has to be substituted with a user-given layer, in order to implement the desired protocol encoding scheme.

Another task that the spi2Java refiner carries out is to statically assign a type to each spi calculus term. This is necessary because the spi calculus is an untyped language, while Java is statically typed. An extensible type hierarchy, reported in figure 3, has been defined for this purpose.

Here is a brief description of the meaning of types shown in figure 3, more detailed information about some specific types is given after this listing.

Message is the most generic type: it represents an opaque message.

Name is a partially specialized type that represents any atomic spi calculus term (i.e. a spi calculus name). Subtypes of this type are used to specialize the meaning of atomic data. Terms belonging to most of the subtypes of the Name type can be instantiated as fresh data.

- **Channel** is the abstract representation of generic communication channels and has some extensions that are worth noting:
  - **Tcp/Ip Channel** provides access to the Tcp/Ip communication layers;
  - **Key Store Channel** provides access to an abstract key store. One of its specialization is the *Java Key Store Channel* that uses the Java key store implementation. Other implementations could be defined, such as an openSSH compatible key store.
  - **File Channel** provides access to the local file system.
  - **Cast Channel** is an artificial channel used to implement Java type casts.
- **Key Pair** represents a pair of public/private keys for use with asymmetric cryptosystems. Note that a key pair is still an atomic message, because public/private keys are obtained by $^+$ and $^-$ constructors, and not by pair splitting.
- **Nonce** represents a randomly chosen sequence of bits.
- **Identifier** represents some information which identifies an entity in a unique way. For example, it can be used as an alias to identify a key or a certificate stored inside a key store. It cannot be instantiated fresh. In order to obtain a random string, the *Nonce* type should be used.
- **Timestamp** represents a time snapshot.
- **Integer** represents an integer number.
  - **Integer With Bounds** represents an integer number that must fit within a given range.

Shared Key represents a key for use with symmetric cryptosystems.

Public Key represents the public component of a key pair used with asymmetric encryption.

- **Certificate** represents a digital certificate. It is considered a specialization of Public Key because each digital certificate contains a public key, along with some other information.

Private Key represents the private component of a key Pair.

Hashing represents the result of applying a non-invertible function on some data.

- **Cryptographic Hashing** represents the result of applying a cryptographic hash function (also known as message digest), such as SHA-256 or MD5, to some data.
- **DH ModPow** represents the result of applying exponentiation $g^x \bmod p$ to its arguments.
  - **DH Pub** represents a Diffie-Hellman public part, obtained by the modular exponentiation of the argument x (private part) with the parameters g and p.
  - **DH Key** represents a Diffie-Hellman shared secret, obtained by the modular exponentiation of the arguments x (private part) and y (other party public part) and parameter p.

Shared Key Ciphered represents the result of a symmetric encryption performed using a Shared key:

Private Key Ciphered represents the result of an asymmetric encryption performed using a Private key.

Digital Signature represents a digital signature, obtained from a Message and a Private Key.

Public Key Ciphered represents the result of an asymmetric encryption performed using a Public key.

Successor represents the logical successor of some data. (Currently only implemented as successor of an *Integer.*)

Pair represents a container of a pair of objects that can be of heterogeneous types. A tuple of objects is translated, inside the program, into nested Pair objects.

It must be pointed out that *Shared Key*, *Private Key* and *Public Key* are not considered atomic names because they are built using the key construction operators from a *Key Pair* or from some key material (e.g. a nonce).

Some detailed information is given on usage of the following types: *Key Store Channel, Cast Channel,* and the *DH ModPow, DH Pub* and *DH Key* hashings.

Key Store Channel Key store channels can be used to save and restore cryptographic keys associated with some alias. In order to interact with the key store through this channel, a send-receive policy is enforced. The user must begin an operation with a `send()` method invocation, and end it by reading its result via a `receive()` method invocation.

In order to get a persistent behavior, closing a key store after it has been used is mandatory, otherwise all newly saved keys will be lost, and deleted keys will not be actually deleted in the backing files.

In general, the argument to the `send()` call will have the form $(OP, \ldots)$, where OP is an operand taken from the `KeyStoreOperand` enumeration.

When the operand is $CHECK$, then the argument of the `send()` method must have the form $(CHECK, LIST)$, where $LIST$ is defined as $LIST := ID|(LIST, ID)$ and where ID is an object of the Identifier class (That is, $LIST$ is a left-associated nested pair of Identifiers). The resulting alias used in the key store will be the concatenation of all provided Identifiers. The $CHECK$ operand asks the key store to check whether any key is associated with the alias LIST. The outcome will be provided by the key store when the `receive()` method will be invoked (more on this later).

With the $CHECK\_IP$ operand, the argument of the `send()` method must have the form $(CHECK\_IP, LISTCH)$, where $LISTCH = CH|(LIST, CH)$, where CH is an object of any of the classes extending Channel that support the `getRemoteURI()` method (e.g. TcpIpChannel). In this case, the checked alias will be the concatenation of the $LIST$ identifiers, concatenated to a string representing the URI of the given channel. This $CHECK$ variant is especially useful when a client wants to deal with server keys, because a server will have a constant and unique URI, which can be used as a key store alias.

When the operand is GET, then the argument of the `send()` method must have the form $(GET, LIST)$. The GET operand asks the key store to retrieve the key associated with the alias $LIST$. The retrieved key will be provided by the key store when the `receive()` method will be invoked, or an exception will be thrown if there is no key associated with the alias $LIST$ (more on this later).

The GET_IP operand has the same meaning of GET, but uses a $LISTCH$ as alias.

If protocol execution can stop immediately if a key is not found in the keystore, then the GET operands can be used directly: if the key is not found an exception will be thrown, and the protocol execution stopped. Otherwise, if different actions should be taken depending whether a key is present or not, one can first use the $CHECK$ operands, and then use the GET operands only if a key was found.

With the PUT operand, the argument of the `send()` must have the form $(PUT, ((LIST, KEY), MODE))$, where KEY is an object of the KeyPair, PublicKey (or Certificate), or SharedKey classes, and $MODE$ is taken from the `KeyStoreMode` enumeration. Currently, $MODE$ is discarded, and the key store behavior is as if $OVERRIDE$ was always given. The PUT operand asks the key store to save the given KEY under the given alias $LIST$. The following `receive()` method invocation will inform the user whether the key was successfully inserted or not.

The PUT_IP operand has the same meaning of PUT, but uses a $LISTCH$ as alias.

After a `send()` method has been invoked, a `receive()` method must be invoked. The argument to be passed to the `receive()` method depends on the previous call to the `send()` one.

If the requested action was a $CHECK$ or $CHECK\_IP$, then the received message will be $OUTCOME$, where $OUTCOME$ is an Identifier storing one of the values of the `KeyStoreOutcome` enumeration. If $OUTCOME$ is $SUCCESS$, then the alias previously given with the $CHECK$ operand is associated with some key, otherwise the outcome will be $FAILURE$.

If the requested action was a GET or GET_IP, then the received message will be KEY, where KEY will be the retrieved key, or an exception will be thrown if no key was associated with the previously given alias.

If the requested action was a PUT or PUT_IP, then the received message will be $OUTCOME$. Currently, as the $OVERRIDE$ mode is the only one implemented, a PUT will always return a $SUCCESS$ status. In future, a PUT action may fail, for example if the key already exists and the $NOT\_OVERRIDE$ mode was requested.

The JavaDoc documentation of the `KeyStoreChannel` provides sample code useful to interact with the key store.

In the running example, the sofA and sofB actors interact with the key store in the way described here, and the key store process models a simple interaction implementing the described policy.

Cast Channel Since spi calculus is untyped, there is no concept of type casts, and any term can be used in any place. For example, the following term $\{M\}_{H(N)}$ that uses an hash as a shared key is a valid spi calculus term. Even the following input process $H(N)(x).P$, which uses an hash as a channel, is a valid spi calculus process. While this is a useful feature of spi calculus, as it enables one to find type flaw attacks, such terms or processes are considered *non well-formed* by the Spi2Java refiner, as two incompatible types should be assigned to the same term. In general, checking for well-formedness is the intended behavior when generating Java code, because only objects of a given type offer the appropriate methods. For example, only channels offer the `receive()` and `send()` methods used to implement the spi calculus input/output processes; an hashing does not offer such methods, and should not be used as a channel.

However, there are circumstances where it is needed to use the same data under two different, incompatible types. One common case is to derive shared secrets from raw data. In this case, cast channels are still not needed. The provided "Shared Key" term does the job, enabling to create shared keys from any other term.

Another common case, where cast channels are needed, is when some non-atomic data should be used as cryptographic parameters of other terms. Suppose a two-party protocol, where both client and server derive a raw shared secret material RAW. Suppose they extract some initialization vector from RAW. In spi calculus, this can be modeled with the $H(RAW, IV\_MARKER)$ term, where IV_MARKER is a marker that signals which part of the RAW material to use. Now, $H(RAW, IV\_MARKER)$ will be typed as $Hashing$, but we would like to use it as the initialization vector of some cryptographic operation, which requires it to be typed as $Identifier$. Cast channels help solving this problem; the following spi calculus snippet shows how to do it:

$$\overline{cast}\langle H(RAW, IV\_MARKER)\rangle.\ cast(IV).\ P$$

That is, one sends the original term to the cast channel, and then reads another term from it. Semantically, it will be the same term, but two different types can be assigned to the terms. So, now the IV term can be typed as `Identifier`, and used as initialization vector for some cryptographic operation. Like the key store channel, a send-receive policy is enforced.

Note that the cast channel will keep the encoded (marshaled) version of the term in memory between the send-receive operations. In particular, it is important that both the original term and the cast term agree on the marshaled representation of data.

As a matter of fact, the default encoding layer using serialization cannot be used with the cast channel, because the marshaled version of the object stores its type, making it impossible to deserialize the object under another type. In order to use cast channels, a custom encoding layer must be provided, which is the code actually implementing the cast logic by the way.

Finally note that cast channels can be used to rename terms too (if the original and final types are the same). In this case, the default encoding layer works too.

DH types The *DH ModPow* is a special hash that accepts one argument of the form $((g,x),p)$, where each of g,x and p are of type *Integer* or *DH ModPow* (or any of their subtypes). The result of the hash is the modular exponentiation $g^x \bmod p$. Along with the equation

$$DHModPow(DHModPow(g,y,p),x,p) = DHModPow(DHModPow(g,x,p),y,p) \tag{1}$$

this type can be used to implement Diffie-Hellman key agreement.

Unfortunately, some protocol analyzers such as ProVerif cannot handle equation (1). For this reason, the two *DH Pub* and *DH Key* types are provided, which work around the issue. The *DH Pub* accepts one argument of the form x, where x is either an *Integer* or a *DH ModPow*. This type has two cryptographic parameters, namely g and p, so that the $g^x \bmod p$ result can be computed.

The *DH Key* type takes one argument of the form (f,x), where each f and x are of type *Integer* or *DH ModPow*. The f value is supposed to be the public part of the other participant, while x is the private part of this participant. This type has one cryptographic parameter, namely p, so that the shared secret $f^x \bmod p$ can be computed.

By using these two types, equation (1) can be rewritten as

$$DHKey(DHPub(y),x) = DHKey(DHPub(x),y)$$

which is supported by protocol analyzers such as ProVerif.

Note that the *DH ModPow* and *DH Pub/DH Key* representations are completely equivalent, because Spi2Java allows cryptographic parameters to be resolved either at compile time, or at run time.

The extensible type hierarchy allows new types to be added when the need arises. For instance, new channel extensions could be defined, in order to provide access to other communication layers, such as UDP/IP, or to other system provided functions, such as databases.

Types are assigned by the Spi2Java refiner using a set of *inference rules*, that, based upon the use of a term in the spi calculus model, assign it the correct type. As stated above, it is possible that the type of a term needs to be manually refined into a more specialized type. However, the Spi2Java refiner checks that the type hierarchy is not infringed. For instance, if a term is automatically typed to *Channel*, it can be manually refined to the *Tcp/Ip Channel* or the *Java Key Store Channel*, but it cannot be typed as *Message* or *Timestamp*.

Furthermore, there is a relationship between the type of a term and its associated low-level parameters. On one hand, each type has its own extensible set of cryptographic and configuration parameters. For instance, the *Shared Key* type has the *algorithm*, *strength* and *provider* parameters, which respectively specify the key cryptographic algorithm, the key strength and the cryptographic provider that will implement the required cryptographic functions. On the

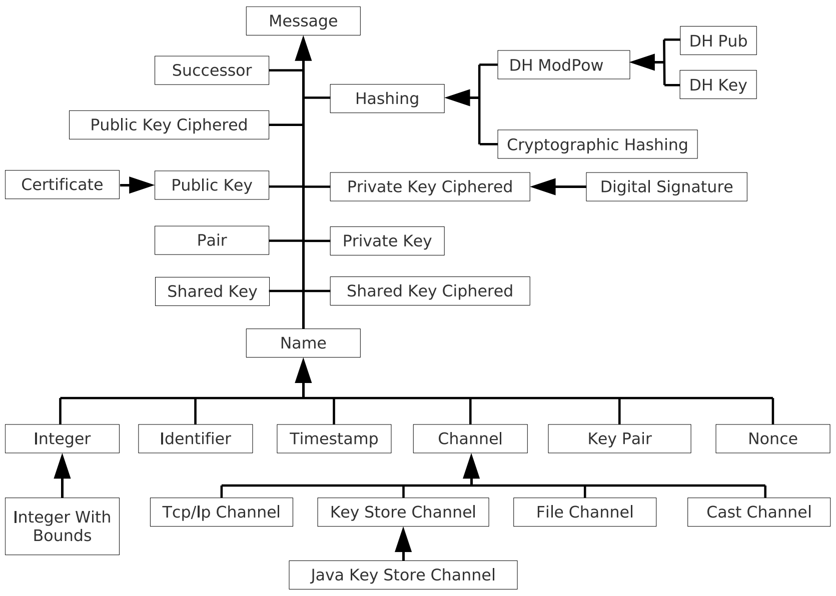

Figure 3: The currently defined eSpi type hierarchy.

other hand, for each type, an extensible set of Java classes can implement the encoding/decoding functions.

In order to store, for each spi calculus term, the assigned type and its low-level implementation details, the Spi2Java refiner uses an XML document, called *eSpi document* (where eSpi stands for "extended Spi"), which is coupled with the original spi calculus source. The eSpi document is automatically generated by the spi2Java refiner, and, when no information can be automatically inferred, default values are used.

In order to let the type hierarchy and the associated parameters be extensible, an XML document, called the *eSpi specification*, which contains these pieces of information, is parsed at run time by the spi2Java refiner; no information is hard coded into the tool. The default encoding layer and default values of parameters are also stored into the eSpi specification, so that they can be modified at any moment too.

For example, the type *Nonce* is represented by the following fragment of the eSpi specification:

```
<type name="Nonce">
  <default_marshall_class name="NonceSR"/>
  <requires>
    <param name="size" default_value="8"/>
  </requires>
</type>
```

A type named "Nonce" is defined, and the class implementing the default encoding layer is called "NonceSR" (more on this below). This type requires one parameter, named "size", whose default value is 8 bytes.

Adding a custom type to the hierarchy is rather straightforward, and is documented in section 4.5.

Moreover, the Spi2Java refiner allows the specification of cryptographic and configuration parameters to be done either statically at compile time, or dynamically at run time. The latter behavior enables the implementation of protocols that use the cryptographic algorithms in the way they have been negotiated at run-time by a cryptographic agreement pre-protocol handshake. In such case, the Spi2Java refiner enables specifying that the value of a certain spi calculus term has to be used as a parameter for a cryptographic operation.

With respect to the data flow diagram in figure 2, the Spi2Java refiner can be used in order to obtain the initial eSpi document coupled with the spi calculus protocol that is going to be implemented.

In particular, the eSpi document for the sofA process can be obtained in two steps.

Step 1 Only the spi calculus model (and implicitly the eSpi specification) are given as input to the spi2Java refiner. The tool generates an eSpi document using all the information that can be automatically inferred from the given model. In particular, types are assigned to terms based on their use, while cryptographic and configuration parameters and the encoding layer are set to their defaults.

Step 2

1. The generated eSpi document is manually refined, adding the needed information that could not be automatically inferred.
2. The spi2Java refiner runs again, taking the manually modified eSpi document and the spi calculus model (and implicitly the eSpi specification) as input. The newly generated eSpi document contains all the information that could be automatically inferred from the model and from the manually provided information.

It must be pointed out that refinement (step 2) can be repeated as many times as needed, until a satisfactory eSpi document is generated by the spi2Java refiner. However, in most cases, like in this example, one run of step 2 is enough.

4.3.2 Practice

Now we will show how the above presented steps can be accomplished in practice.

Step 1: Creating an eSpi Template.
The eSpi document for the sofA process can be created by using the `SpiTyperTextual` program, included in the `typer.tui.jar` archive.

So, to create a first eSpi document from the symbol table of the sofA process the following command can be used:

```
java -jar $JARS/typer.tui.jar -s spi/sofA/sofA.css
          -o spi/sofA/sofATemplate.xml
```

As a result, the eSpi document is created as "`sofATemplate.xml`" (since `-o` option specifies this as the output file), from the symbol table file (specified by the `-s` option). The optional `-f` option can be supplied to instruct the program to silently overwrite the output file if this already exists.

Note that the default eSpi specification will be used by the program. If you added custom types to the eSpi specification, you can instruct the program to use your custom eSpi specification by using the `-e` option.

In the eSpi document, a `term` element contains additional information about a spi calculus term declared in the formal model. Three attributes are present in this element: `id` is a unique identifier for the term, used internally by the spi2Java tools; `name` is a human readable representation of the term, useful in order to uniquely identify it when manually modifying the eSpi document; `type` contains the information about the type that has been statically assigned to the current term. The value of the `type` attribute must be present in the eSpi specification, and must be coherent with the use of the term in the model, as inferred by the spi2Java refiner. The `codify` element contains the name of a Java class implementing the encoding layer for the current term.

Note that the eSpi document refers to the expanded form of the input description, where for example sequences of terms of length greater than 2 are transformed into nested pairs. For this reason, it is possible that some new terms, not present in the original specification, appear in the eSpi document. In order to understand the meaning of these terms, it is possible to look at the expanded form of the input specification, that can be generated by using the `-r` when compiling the spi calculus sources with the Spi2Java parser.

Step 2.1: Refining the eSpi document.
Once the `sofATemplate.xml` document has been created, there may be the need of specializing the type of some terms.

First, all channels and hashings must be manually specialized to any of their subtypes, as the *Channel* and *Hashing* types are abstract types, that cannot be instantiated. For this purpose, the dedicated *typer-by-default* tool is available, which downcasts all abstract types to a default concrete type. The following command

```
java -jar $JARS/typerByDefault.tui.jar -s spi/sofA/sofA.css
          -x spi/sofA/sofATemplate.xml
          -o spi/sofA/sofATemplateWithDefault.xml
```

creates the `sofATemplateWithDefault.xml` eSpi document, where all abstract channels are downcast to *Tcp/Ip Channel*, and all abstract hashes to *Cryptographic Hashing*. Also note that the default downcast rules for the typer-by-default are fully customizable, through command line parameters. See section 5.1 for more information about the tool.

Now, the obtained eSpi document can be manually modified. First of all, before modifying the file, it can be useful to create a sandbox copy of `sofATemplateWithDefault.xml` that we call `sofAEdited.xml`.

In this specific case the terms to be refined are:

- terms 0, 4, 7: downcast to the *Identifier* type (and updating the encoding layer accordingly).
- term 1: specify that this channel will be a server channel, using the "TcpIpServer" implementation (more on this later).
- term 2: change from the default *Tcp/Ip Channel*, to the *Java Key Store Channel*, and updating the encoding layer and parameters accordingly.
- terms 5, 11, 12: downcast to *Nonce*, and update encoding layer and parameters accordingly.

- terms 10, 18: change cryptographic algorithm and strength. In order to comply with the protocol specification, when using the pre-shared session key, we must use 3DES, with strength of 192 bits. Since no initialization vector has been prescribed, we can use the constant default one.
- term 16: the agreed session key is a protocol return value: let this value be available to the caller of this protocol run, by setting the `return` attribute to `true`. In an eSpi document, the `return` attribute can be set in as many terms as needed: the Spi2Java code generator will take care of making all protocol return parameters available to the Java code that is executed after a successful protocol session.

An important difference between the sofA and sofB roles concerns the communication channel: sofA must behave as a responder (or server), while sofB must behave as an initiator (or client). Therefore, in the sofA case, the `server` attribute is set for the term 1, *cAB*_0, to specify that the process owning this term will act as a responder on the *cAB*_0 channel (using the responder channel implementation provided in the *TcpIpServer* Java class), so the Spi2Java code generator will take this into account. Since no term in the sofB process has been manually given a `server` attribute, it will be implemented as an initiator. Only when a channel is set as "server", it is possible to specify the keyword "any" for the `host` parameter, letting the server listen on any available network interface. In each eSpi document, at most one channel can be set as the server channel.

When casting term 2 to be a key store, the user must specify two files that will contain persistent key store data, associating stored keys with their identifiers. For the reference example, these two files, namely `sof.keystore` and `sof.keystore.data`, are provided, which are already filled with the pre-shared 3DES key, associated with the "AB" identifier. Unfortunately, the current version of Spi2Java does not provide any user interface to handle the key store data files. However, a key store and its related files can be managed programmatically by interacting with channels, as any protocol actor implementation would do. The program written for the purpose of filling the `sof.keystore` and `sof.keystore.data` files is provided with the reference example, under the `java/fillKS` directory. Please remember that there is no need to run this tool, as its resulting files are already provided. Anyway, for the sake of completeness, to compile the tool, run:

```
javac -cp java/fillKS:$JARS/spiWrapper.jar:$JARS/spiWrapperSR.jar
      java/fillKS/*.java java/customMarsh/*.java
```

To run the tool:

```
java -cp java:$JARS/spiWrapper.jar:$JARS/spiWrapperSR.jar
     fillKS.FillKS
```

If "SUCCESS" is printed on stdout, then everything is fine, and the `sof.keystore` and `sof.keystore.data` files are created in the current directory. Otherwise "FAILURE" is printed (should not happen).

When refining term 16, the class implementing the encoding layer was changed to a non default encoding layer. Please ignore this detail by now. Implementing a custom encoding layer is explained in section 4.4.2.

Each `param` element has an attribute called `type`, which is used in order to specify whether the parameter is assigned at compile time (`type="const"`) or it must be resolved at run time (`type="var"`). If the parameter must be resolved at run time, then its value must be the numeric identifier of another term that will contain the value of the parameter at run-time. When a term is used as an input parameter, the Spi2Java refiner automatically forces its type to *Identifier*. For example, the following snippet is semantically correct:

```
<term id="0" name="alg" type="Identifier">
  <codify>IdentifierSR</codify>
</term>
<term id="1" name="H(data)" type="Cryptographic Hashing">
  <codify>CryptoHashingSR</codify>
  <parameters>
    <param name="algorithm" type="var">0</param>
    <param name="provider" type="const">SUN</param>
  </parameters>
</term>
```

It states that term 1, which is a cryptographic hashing, will use at run time the content of term 0 as the name of the hash algorithm to be used. Conversely, the cryptographic provider is chosen to be "SUN" statically at compile time. Note that term 0 will be forced to be an *Identifier* by the Spi2Java refiner, so trying to change its type will result in an error in step 2.2. If a different type should be assigned to `alg`, then the cast channel can be used.

The *provider* parameter accepts a semicolon separated list of allowed JCA providers, ordered by preference. The most preferred provider supporting the requested algorithm will be chosen. For example, given two providers "provA" and "provB", the following line

```
<param name="provider" type="const">provA; provB</param>
```

asks to use "provA", if it supports the algorithm to be implemented (and it is installed in the Java Virtual Machine being used); otherwise, "provB" will be chosen. If no listed provider supports the requested algorithm, an exception will be thrown at run time.

Remark: It may be argued that manual modification of the eSpi document is not very user friendly. This is true; however the whole spi2Java tool, which is currently a prototype, has been designed to be an integrated development environment (IDE). In this context, a convenient user interface could accept user input, and then could transparently handle XML documents, automatically filling default values or adding required elements, as defined in the eSpi specification. With this design in mind, the use of the machine readable XML document format, and the definition of the eSpi specification get even more importance. For example, when refining the type of a term from *Message* to *Nonce*, the IDE, according to the eSpi specification, can automatically change the default encoding layer, and can add all the required parameters for the new type, filling them with default values, or asking for custom values.

Step 2.2: Using the refined eSpi document.
Now that the `sofAEdited.xml` file has been modified, the *SpiTyperTextual* program has to be run again to check that the file does not contain any syntactic error and that there are no refinements that are not admissible by the Java class hierarchy. Moreover, the refiner could exploit the added information in order to further refine other terms. The *SpiTyperTextual* can be invoked in this way:

```
java -jar $JARS/typer.tui.jar -s spi/sofA/sofA.css
                              -x spi/sofA/sofAEdited.xml
                              -o spi/sofA/sofAFinal.xml
```

It is worth noting that now the `sofAEdited.xml` file is provided to the tool too (by the `-x` option), and a new output file is being specified (i.e. `sofAFinal.xml`). In this case it may be useful to perform a diff between the `sofAEdited.xml` and the `sofAFinal.xml` files to see if there are some differences, i.e. if some terms have been further refined. In this case, no terms have been further refined.

Since there are no more terms to refine, it is possible to proceed with code generation. Anyway, should the specification be not yet complete, step 2 can be repeated until a satisfactory specification is created.

4.4 Implementing the Java Application

Once the final eSpi documents of the prototype version are done, the Spi2Java code generator is used in order to obtain a Java implementation of the given spi calculus model, refined with the information contained in the coupled eSpi document.

The generated code uses classes and methods provided by the *spiWrapper* Java library (previously called *secureClasses* in [7]).

It is worth noting that the spi2Java code generator does not only generate the security critical Java code implementing the spi calculus model; instead, it also generates complete application templates. By initializing some values, these applications can be compiled and executed. This strongly reduces user interaction, enabling agile prototyping.

The Spi2Java code generator can currently implement applications using two different architectures, whose flowcharts are reported in figure 4. If an application must act as an initiator, that is, no `term` element has the `server` attribute in the eSpi document, then the Spi2Java code generator automatically uses the client architecture. Otherwise the application must act as a responder on the channel having the `server` attribute, and the Spi2Java code generator automatically uses the concurrent server architecture. The Spi2Java code generator is designed so that it is easy to add new implementation architectures, such as concurrent crew servers (also known as "prefork") and the like.

It is worth noting that, in the application templates currently provided, while the `performHandshake()` method implements the logic of a protocol session, the `act()` method is executed only if the current protocol session ends successfully (that is, `performHandshake()` returns and does not throw an exception), and is initially empty. This method can be implemented by the user in order to perform any action that must be done after a successful end of the protocol session.

It must be also pointed out that, although the Spi2Java code generator always generates code that can be compiled and executed without any modification, the input parameters of the `performHandshake()` method must be manually initialized before the program can be correctly executed, because no information can be automatically inferred on their contents. The method input parameters are all the free variables that are used in the process being implemented, with the exception of channels, that get configured automatically because the eSpi document already contains enough information for their

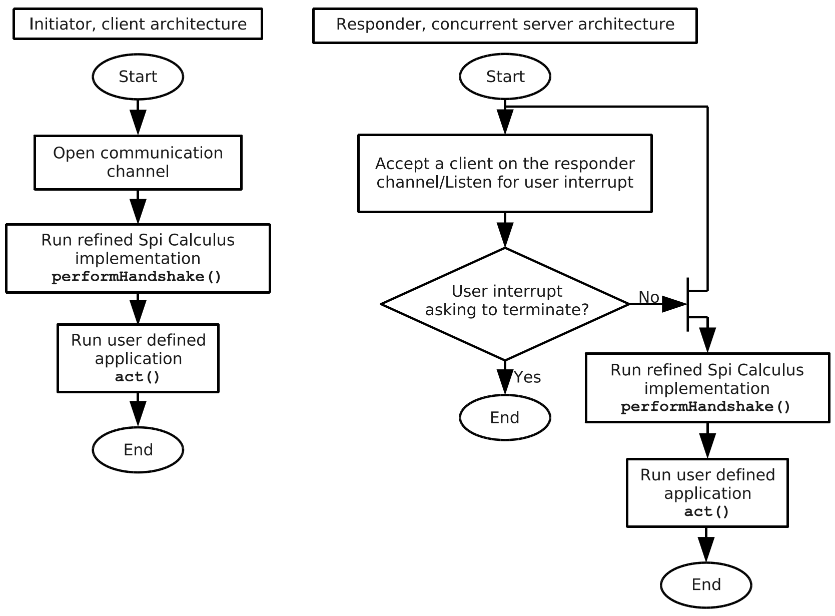
Initiator, client architecture
Start
Open communication channel
Run refined Spi Calculus implementation performHandshake()
Run user defined application act()
End
Responder, concurrent server architecture
Start
Accept a client on the responder channel/Listen for user interrupt
User interrupt asking to terminate?
No
Yes
End
Run refined Spi Calculus implementation performHandshake()
Run user defined application act()
End

Figure 4: Flowchart of the initiator and responder architectures.

initialization. For this reason, in the sof protocol example, variables A and $KEYSTORE\_GET$ must be initialized in the sofA process. It is worth noting that input parameters can be initialized at compile time, or at run time, for example based upon user input.

4.4.1 Obtaining a protocol prototype implementation

When the eSpi specification file is in its final version, code generation can take place. The *JavaGenerator* program can be used to complete this task.

It is now shown how to get an early prototype of the protocol implementation, where the default encoding layer is used. This means that messages are encoded using the Java object serialization mechanism. Later on, it will be shown how a custom encoding layer can be implemented. In order to generate the code that uses the default encoding layer, the *JavaGenerator* program has to be fed with the name of the packages (i.e. it.polito.spi2java.spiWrapperSR and inner packages) that contain the implementation of the encoding layer. This can be done by using the `-i` option, that specifies the name of packages that will be imported by the generated code. If many packages need to be imported, their names can be written in a text file, one per line, and the text file is passed to the *JavaGenerator* program by the `-j` option.

The *JavaGenerator* program can be invoked by using the following command:

```
java -jar $JARS/javaGenerator.tui.jar -o java/generated/fullExample/sofA
          -p fullExample.sofA -j java/imports -s spi/sofA/sofA.css
          -x spi/sofA/sofAFinal.xml
```

An explanation of the used options follows. The `-j` option is used to import the set of packages required to use the encoding layers; all the required imports are specified in the `java/imports` file. The `-o` option specifies the path where the tool must place generated files and the `-p` option specifies the name of the package for the generated code. Note that the user must specify the output directory including the path corresponding to the package name, so it is the user's responsibility to ensure that the last part of the output directory path matches the package name. The `-s` and `-x` options specify the symbol table file and its coupled eSpi document respectively, from which the code is generated. Note that a default eSpi specification and default template files for code generation are used. A custom eSpi specification can be specified by the `-e` option; this option is needed when custom types have been added, and so a custom eSpi specification must be used by the program. Custom template files for code generation can be provided too, via the `-t` option. In this case, the argument to the option must be a directory, which will contain the set of template files. This option is usually not needed for most users.

The Java code generator produces (in the `java/generated/fullExample/sofA` directory specified on the command line) the following files for the sofA server implementation:

`sofA_0_Main.java` : contains the main function starting the server process, listening for clients.

`sofA_0_Callback.java` : contains the code that is called each time a client request has been accepted. This code initializes the parameters of the protocol and invokes the protocol and then the application.

`sofA_0_Protocol.java` : contains the implementation of a protocol session.

`sofA_0_Application.java` : contains the skeleton of an application that is invoked after each protocol session execution.

For the sofB client implementation, the following files are generated:

`sofB_0_Main.java` : contains the main that initializes the parameters of the protocol and that invokes the protocol and then the application.

`sofB_0_Protocol.java` : contains the implementation of a protocol session.

`sofB_0_Application.java` : contains the skeleton of the application that is invoked after each protocol session execution.

It is worth noting that the Java code generator does not know what are the values of the protocol parameters and, hence, their initialization must be manually provided. Therefore, if you try to run the generated code without providing parameters, the generated code execution stops, even before the code actually implementing the spi calculus model is run. So, in order to initialize the parameters, with respect to the sofA actor, the `sofA_0_Callback.java` file needs to be modified.

First, a backup copy of `sofA_0_Callback.java` is created into `sofA_0_Callback.java.orig`. Then the generated file `sofA_0_Callback.java` is modified such that the following lines

```
Identifier A_0 = null;
Identifier KEYSTORE_GET_0 = null;
```

become

```
Identifier A_0 = new IdentifierSR("A");
Identifier KEYSTORE_GET_0 =
                    new IdentifierSR(KeyStoreOperand.GET.operand());
```

For convenience, the `sofA_0_Callback.java.edited` file contains the Java source code in its final state, with manual modifications, while the plain generated source code can be found in `sofA_0_Callback.java.orig`. The `sofA_0_Callback.java` file distributed with Spi2Java is already in its final state, but it will be overwritten if the code is generated again.

Note that in the sofB client process similar initializations must be done, but on the `sofB_0_Main.java` file, because it is a client implementation. Again, `.edited` and `.orig` files are included for convenience, and the distributed version of the `sofB_0_Main.java` file is already in its final state.

It is worth noting that the `new` operators used in these statements refer to target classes that provide concrete data types, i.e. classes that include the implementation of an encoding layer (e.g. the *IdentifierSR*). Classes belonging to the *it.polito.spiWrapper* package, such as *Message*, cannot be instantiated, because they are abstract classes. In this case, the classes using Java Serialization as the default encoding layer have been used (i.e. classes belonging to the *it.polito.spiWrapperSR* package). The names of these classes slightly differ from those included in the *it.polito.spiWrapper* package, because their name ends with the `SR` post-fix.

Finally, the sofA actor can be compiled and ran by the following commands.

Compiling:

```
javac -cp $JARS/spiWrapper.jar:$JARS/spiWrapperSR.jar
      java/generated/fullExample/sofA/*.java java/customMarsh/*.java
```

Running:

```
java -cp $JARS/spiWrapper.jar:$JARS/spiWrapperSR.jar:java/generated:java
     fullExample.sofA.sofA_0_Main
```

And similar can be done for the sofB client.

Let us now comment on how the protocol logic expressed in spi calculus translates into Java code. For example, please look at the generated `sofA_0_Protocol.java` file. It is worth noting that the Spi2Java code generator automatically adds comments to the code. So, they both improve code readability and make clear the mapping between each spi calculus statement and its Java implementation. In particular, it can be noted that each spi calculus statement is mapped on a few corresponding Java statements, and all the operations on a spi calculus term are handled by methods of the spiWrapper class implementing the type assigned to the term.

By now, although the used encoding scheme may not be compliant with the protocol description, the prototype programs are fully implementing the protocol logic, so they can be used to test the protocol behavior and functionality. This test step is important, because a protocol could have been formally verified and resulted to be safe, even if it is not functional, and thus useless. For example, suppose a protocol where all sessions abort at the very beginning, because of a wrong design. Then this protocol is probably satisfying secrecy (and possibly authentication) requirements, because the intruder cannot get the secret, since no message is ever exchanged. However, such a protocol is useless.

Once the protocol functionality has been tested on the prototype applications, the encoding layer can be implemented, in order to obtain fully functional and interoperable applications.

4.4.2 Customizing the Encoding Layer

In order to create the encoding layer, four abstract methods declared in the spiWrapper classes must be implemented by the programmer for each type of encoding that is required by the specification. More precisely, the four methods can be implemented by extending the spiWrapper class representing the type for which the encoding scheme is going to be written. It is worth noting that this approach isolates hand-written code with respect to automatically generated code.

The four methods that must be implemented are:

- `_encodePayload();`
- `_serialize();`
- `decodePayload();`
- `deSerialize().`

Note that `_encodePayload()` and `_serialize()` begin with a leading `_` because of legacy reasons.

The first method is responsible for translating the internal representation of a term into the payload. It is used when some cryptographic operations (such as encryption or hasing) must be applied on the term.

The second method is used to serialize the term into a message, i.e. a byte stream including payload and any needed headers and trailers, ready to be sent on a communication channel. This is the classical marshaling needed by communication protocols.

This approach gives high flexibility, by allowing different and independent encodings for cryptographic and networking operations.

The third and fourth methods are dual with respect to the first and second ones. `decodePayload()` transforms the encoded payload into the internal representation of the term, while `deSerialize()` extracts the payload from a received message coming from a communication channel.

The class `NonceRaw.java` provided in the `java/customMarsh` directory is an example of how a custom encoding layer can be implemented. When encoded, this nonce just outputs its raw bytes, with no header or trailer. It is used as key material when generating the shared session key in the protocol. The shared session key constructor uses the first 8 bytes of the key material, that are taken by the encoded nonce. By not using a fixed header or trailer in the nonce encoding, it is avoided that the same constant header data are always used as key material in each protocol run.

In the `NonceRaw` class, deserialization and decoding have not been implemented as they are not necessary: no raw nonces are ever read from the network or decoded as a result of cryptographic operations. If needed, they could be implemented as the inverse of their respective serialization and encoding functions.

4.5 Adding Custom eSpi Types

Adding custom types to the eSpi types hierarchy is rather straightforward. First get a copy of the `espi_specification.xml` file (which is located in the `data/eSpi` directory from the root of the released framework).

Then, suppose the type "UDP/IP Channel" must be added, as a subtype of "Channel". Within the `type` element representing the "Channel" type, add a nested `type` element representing the "UDP/IP Channel" type. This new type element could look like

```
<type name="UDP/IP Channel">
  <default_marshall_class name="UDPIPChannelSR"/>
  <requires>
    <param name="host" default_value="localhost"/>
    <param name="service" default_value="2006"/>
    <param name="timeout" default_value="0"/>
  </requires>
</type>
```

Afterwards, create the `abstract class UDPIPChannel extends Channel` in the `it.polito.spi2java.spiWrapper` package and implement the custom logic for this type. The name of the class can actually be arbitrary, but it is a best practice to name it after the type name, with spaces and special characters removed. Note that the `it.polito.spi2java.spiWrapper` package can be a package local to your project, there is no need to add the new class into the Spi2Java package with the same name. The Java compiler and run time tools will be able to handle dependencies correctly. Also note that all abstract methods of the supertypes, except the four methods of the encoding layer, are supposed to be implemented.

Finally, create the `class UDPIPChannelSR extends UDPIPChannel` in the `it.polito.spi2java.spiWrapperSR` package, and implement the remaining four methods of the encoding layer, by using serialization.

When using the Spi2Java tools, if custom types are added, always remember to use the custom eSpi specification file, by specifying the proper option on the command line. Otherwise, the default eSpi specification will be used, which does not contain information about the custom types, and errors will be reported.

5 Spi2Java Auxiliary Tools

This section enumerates and briefly describes some auxiliary tools that are included in the Spi2Java framework. For a reference of all options available in each tool (included the main tools explained above), please refer to the online help.

5.1 Typer by Default

Some eSpi types, for instance *Channel* or *Hashing*, are abstract, and cannot be instantiated. Moreover, when automatically inferring types, it is not possible to choose an appropriate concrete type for these abstract types. When such abstract types are inferred, manual refinement becomes necessary in order to let the code compile and execute. However, if agile prototyping is aimed to, or all abstract types would be refined to the same concrete type, then it is possible to use the *TyperByDefault* program, that automatically refines each abstract type to a default concrete type.

The mapping between each abstract type and its default concrete type can be specified via the command line, by the `-m` switch, or through a configuration file, specified via the `-n` switch. The configuration file must contain one mapping per line; everything after a `#` is a comment. The following is the content of an example mapping file:

```
# This is a comment line
# The format is "AbstractType:DefaultConcreteType"
Channel:Tcp/Ip Channel
Hashing:Cryptographic Hashing
```

A usage example of the tool is presented when refining the reference example.

5.2 eSpi Merger

Sometimes it happens that, when manual refinement of the eSpi document is almost complete, one discovers some errors in the protocol specification. Fixing the protocol leads to a new coupled eSpi document, which must be manually refined from scratch again.

In order to reduce the impact of this issue, the *eSpiMerger* program can be used. It accepts two files, a *new* file and an *old* file. The *new* file is the eSpi document obtained from the latest protocol specification, thus containing up-to-date but still unrefined terms. The *old* file is the eSpi document obtained from the previous protocol specification, thus containing not up-to-date, but refined terms. The *eSpiMerger* program will try to refine the terms in the *new* file, by looking at how similar terms in the *old* file had been refined.

For example, suppose a flawed specification of the sofA actor was given, and that it had been refined up to the point that a complete `sofAFinal.xml` eSpi document was ready.

At some point, the flaw is discovered, and the sofA specification is updated. This leads to a new coupled `sofATemplate.xml`, which must be refined from scratch. The *eSpiMerger* program can be invoked like this:

```
java -jar $JARS/espiMerger.tui.jar -a /spi/sofA/sofATemplate.xml
                                   -b /spi/sofA/sofAFinal.xml
                                   -o /spi/sofA/sofAMerged.xml
```

where `-a` indicates the *new* file, `-b` the *old* file, and `-o` the resulting output file.

As a best practice, the `sofAMerged.xml` file can then be copied in a file called `sofAEdited.xml`, and the normal manual refining steps can be performed.

5.3 Spi2Proverif

The Spi2Proverif auxiliary tool translates a spi calculus specification expressed in the syntax accepted by the Spi2Java framework, into a semantically equivalent specification expressed in the syntax accepted by the ProVerif verification tool. This allows the Spi2Java users to verify their models with the ProVerif analyzer [2].

Two things must be noted about the translation.

- As the current version of Spi2Java does not support replication, the translated process will not include replication either; it is needed to add replication operators manually in the generated ProVerif specification.
- A Spi2Java protocol specification only describes the protocol behavior; it does not contain any information about the intended security properties. For this reason, the translated ProVerif specification does not include any security query, nor protocol execution events. They have to be manually added to the ProVerif specification, before the analysis of the protocol.

The reference "sof" protocol has been verified for secrecy and agreement by using the ProVerif tool, and all related files are under the `proverif` directory. While usage of the ProVerif tool is outside the scope of this document, the steps to translate the protocol specification to the ProVerif format and to decorate it with the security queries are described.

The sof protocol can be translated to the ProVerif syntax with the following command:

```
java -jar $JARS/spi2proverif.tui.jar -i spi/sof.spi -o proverif/sof.proverif
```

Then, the generated `sof.proverif` file is copied into the `sof.queries.proverif` file, where the replication operators and the security queries are added.

Finally, the ProVerif analyzer is run on the `sof.queries.proverif` file.

Note that the top level process (i.e. the *Inst* process) has been translated to ProVerif. Indeed, it is necessary to translate the whole protocol description, and not, for example, single actors, to verify the whole protocol behavior.

References

[1] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: The spi calculus," in *ACM Conference on Computer and Communications Security*, 1997, pp. 36–47.

[2] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Computer Security Foundations Workshop*, 2001, pp. 82–96.

[3] L. Durante, R. Sisto, and A. Valenzano, "Automatic testing equivalence verification of spi calculus specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 12, no. 2, pp. 222–284, 2003.

[4] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, parts I and II," *Information and Computation*, vol. 100, no. 1, pp. 1–77, 1992.

[5] J. Clark and J. Jacob, "A survey of authentication protocol literature: Version 1.0," 1997.

[6] A. Pironti and R. Sisto, "An experiment in interoperable cryptographic protocol implementation using automatic code generation," in *IEEE Int. Symp. on Computers and Communications*, 2007, pp. 839–844.

[7] D. Pozza, R. Sisto, and L. Durante, “Spi2java: Automatic cryptographic protocol java code generation from spi calculus,” in *International Conference on Advanced Information Networking and Applications*, 2004, pp. 400–405.